

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЁТ
по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: Рекурсивная обработка иерархических списков

Студент гр. 8304		Ястребов И.М.
Преподаватель		Фирсов М.А.

Санкт-Петербург
2019

Задание.

Вариант №29

Задания про иерархическое содержание.

Иерархическое содержание состоит из записей, каждая запись имеет иерархический номер, название и содержит либо несколько подчинённых записей ("организующая запись"), либо текст ("текстовая запись"). Пример иерархического содержания:

0 Предисловие [текст предисловия]

1 Книга I

1.1 Часть 1

1.1.0 Пролог [текст пролога]

1.1.1 Глава 1 [текст главы]

1.1.2 Глава 2 [текст главы]

1.1.3 Глава 3 [текст главы]

1.1.4 Глава 4 [текст главы]

1.1.5 Эпилог [текст эпилога]

1.2 Часть 2

1.2.0 Пролог [текст пролога]

1.2.1 Глава 1 [текст главы]

1.2.2 Глава 2 [текст главы]

1.2.3 Глава 3 [текст главы]

1.2.4 Эпилог [текст эпилога]

2 Книга II

2.1 Глава 1 [текст главы]

2.2 Глава 2 [текст главы]

2.3 Глава 3 [текст главы]

2.4 Глава 4 [текст главы]

2.5 Эпилог [текст эпилога]

3 Книга III

3.-1 Предисловие к книге III [текст предисловия]

3.0 Пролог [текст пролога]

3.1 Часть 1

3.1.1 Глава 1 [текст главы]

3.1.2 Глава 2

3.1.2.1 Подглава 1 [текст подглавы]

3.1.2.2 Подглава 2 [текст подглавы]

3.1.3 Глава 3 [текст главы]

3.2 Отступление между частями 1 и 2 [текст отступления]

3.3 Часть 2

3.3.1 Глава 1 [текст главы]

3.3.2 Глава 2 [текст главы]

3.3.4 Главы 3-4 [текст глав]

3.4 Части с утерянным текстом

3.7 Эпилог [текст эпилога]

4 Послесловие [текст послесловия]

В текстовой записи "3.3.4 Главы 3-4 [текст глав]" "3.3.4" - это иерархический номер записи, "4" - относительный номер записи, "Главы 3-4" - название записи, а в квадратных скобках приведён текст записи.

Иерархическое содержание может быть представлено иерархическим списком, каждый элемент которого имеет название и данные - список подчинённых записей либо текст. Позиция записи в иерархическом списке определяется её номером. В элементе иерархического списка следует хранить относительный номер записи, а не её полный иерархический номер. Относительный номер каждого последующего элемента в списке должен быть больше, чем у предыдущего.

Запись с номером a.b.c обязательно должна быть в списке подчинённых записей элемента a.b (не в списке элемента a).

Задания:

Разработать программу, создающую иерархический список, соответствующий иерархическому содержанию, записанному в файле в приведённом выше формате. Записи могут быть в файле перепутаны, но все имеют различные номера. Программа должна позволять выводить иерархическое содержание на экран или в файл (в правильном порядке), добавлять новые записи (если запись с заданным номером уже существует, то сообщать об ошибке), а также выполнять дополнительные действия, в зависимости от варианта:

29)

- Нормализовать номера элементов списка (элементы списка нумеруются натуральными числами, начиная с 1, без пропусков); операция нормализации может быть применена к верхнему уровню иерархического содержания или к подписку с заданным иерархическим номером; может затрагивать только указанный список или распространяться на всех его потомков (по выбору пользователя);

- Вставить новую запись (если запись с заданным номером уже существует, то её номер инкрементируется, далее при необходимости по цепочке инкрементируются номера следующих записей).

Цель работы.

Решить полученную задачу, используя иерархические списки. Получение навыков работы с нелинейными структурами данных.

Описание алгоритма.

Считываем из входного файла изначальные данные. Формируем на их основе начальный иерархический список. Далее с помощью командной строки спрашиваем у пользователя, какое действие необходимо выполнить

Описание функций программы:

1) `void load_h_content(std::vector<std::string> &sorted_nodes);`

Функция предназначена для инициализации исходного списка с помощью исходных данных.

`sorted_nodes` – исходные данные

2) `void print(std::shared_ptr<Node> head, std::string index_str);`

Функция предназначена для вывода содержимого списка, `head` – начало списка, `index_str` – наращиваемый префикс

3) `std::shared_ptr<Node> add(std::string &source_str);`

Функция предназначена для добавления записи в список
`source_str` – строка, которую нужно превратить в запись

4) `std::shared_ptr<Node> insert(std::string &source_str);`

Функция предназначена для вставки записи в список
`source_std` – строка, которую нужно превратить в запись

5) `void normalize(const std::string &start, bool recursive);`

Функция предназначена для нормализации списка

5) `void normalize_row(std::shared_ptr<Node> start, bool recursive = false);`

Функция предназначена для нормализации одного уровня списка (с

возможностью рекурсивной нормализации вложенных списков)

- 6) `std::shared_ptr<Node> parse_str_into_node(std::string &source_str);`
Функция предназначена для парсинга строки в структуру узла списка

Описание структур данных:

```
1) typedef struct Node {
    int h_index; // "иерархический номер" (с)

    std::string title; // "название" (с)

    std::variant < std::shared_ptr<Node>, std::string > list_or_text; //
    "либо несколько подчиненных записей, либо текст" (с)

    std::shared_ptr<Node> next;
}Node;
```

Структура предназначена для хранения узла списка

`h_index` – иерархический относительный индекс

`title` – название

`list_or_text` – указатель на начало вложенного списка либо текст

`next` – указатель на следующий элемент в списке

Выводы.

В ходе выполнения работы были изучены нелинейные структуры данных, был получен опыт работы с иерархическими списками.

Протокол

Тестирование:

Входной файл :

0 Предисловие [текст предисловия]

1 Книга I

1.1 Часть 1

1.1.0 Пролог [текст пролога]

1.1.1 Глава 1 [текст главы]

1.1.2 Глава 2 [текст главы]

1.1.3 Глава 3 [текст главы]

- 1.1.4 Глава 4 [текст главы]
- 1.1.5 Эпилог [текст эпилога]
- 1.2 Часть 2
 - 1.2.0 Пролог [текст пролога]
 - 1.2.1 Глава 1 [текст главы]
 - 1.2.2 Глава 2 [текст главы]
 - 1.2.3 Глава 3 [текст главы]
- 2.1 Глава 1 [текст главы]
- 2.2 Глава 2 [текст главы]
- 2.3 Глава 3 [текст главы]
- 2.4 Глава 4 [текст главы]
- 2.5 Эпилог [текст эпилога]
- 3 Книга III
 - 3.-1 Предисловие к книге III [текст предисловия]
 - 3.0 Пролог [текст пролога]
 - 3.1 Часть 1
 - 3.1.1 Глава 1 [текст главы]
 - 3.1.2 Глава 2
 - 3.1.2.1 Подглава 1 [текст подглавы]
 - 3.1.2.2 Подглава 2 [текст подглавы]
 - 3.1.3 Глава 3 [текст главы]
 - 3.2 Отступление между частями 1 и 2 [текст отступления]
 - 3.3 Часть 2
 - 3.3.1 Глава 1 [текст главы]
 - 3.3.2 Глава 2 [текст главы]
 - 3.3.4 Главы 3-4 [текст глав]
 - 3.4 Части с утерянным текстом
 - 3.7 Эпилог [текст эпилога]
- 4 Послесловие [текст послесловия]
- 1.2.4 Эпилог [текст эпилога]
- 2 Книга II

Результат работы программы:


```
.2 Книга II
.2.1 Глава 1 [текст главы]
.2.2 Глава 2 [текст главы]
.2.3 Глава 3 [текст главы]
.2.4 Глава 4 [текст главы]
.2.5 Эпилог [текст эпилога]
.3 Книга III
.3.-1 Предисловие к книге III [текст предисловия]
.3.0 Пролог [текст пролога]
.3.1 Часть 1
.3.1.1 Глава 1 [текст главы]
.3.1.2 Глава 2
.3.1.2.1 Подглава 1 [текст подглавы]
.3.1.2.2 Подглава 2 [текст подглавы]
.3.1.3 Глава 3 [текст главы]
.3.2 Отступление между частями 1 и 2 [текст отступ]
.3.3 Часть 2
.3.3.1 Глава 1 [текст главы]
.3.3.2 Глава 2 [текст главы]
.3.3.4 Главы 3-4 [текст глав]
.3.4 Части с утерянным текстом
.3.7 Эпилог [текст эпилога]
.4 Послесловие [текст послесловия]

Choose action :
1 : print
2 : add
3 : insert
4 : normalize
```

Исходный код

lab2.cpp

```
#include "pch.h"
#include "h_content.h"

int main(int argc, char* argv[])
{
    setlocale(LC_ALL, "Russian");

    std::vector<std::string> input_nodes;

    std::ifstream input;

    input.open("input.txt");

    if (!input) {
        std::cout << "Couldn't open source file";
        exit(1);
    }

    char tmp;
    std::string processed_string = "";

    while (input.get(tmp))
    {
        if (tmp != '\n')
            processed_string += tmp;
        else {
            if (processed_string.size())
```

```

        input_nodes.push_back(processed_string);
        processed_string = "";
    }
}

input_nodes.push_back(processed_string);

input.close();

std::sort(input_nodes.begin(), input_nodes.end());

h_content hList;
hList.load_h_content(input_nodes);
while (true) {
    std::cout << "_____ " << std::endl;
    std::cout <<
        "Choose action :\n1 : print\n2 : add\n3 : insert\n4 : normalize"
        << std::endl;

    int action;

    std::cin >> action;
    std::cin.get();

    if (action == 1)
    {
        hList.print(hList.head, "");
    }

    if (action == 2) {
        std::cout << "enter new node" << std::endl;

        std::string tmp;

        std::getline(std::cin, tmp);

        if (!hList.add(tmp))
            std::cout << "Failed" << std::endl;
    }

    if (action == 3) {
        std::cout << "enter new node" << std::endl;

        std::string tmp;

        std::getline(std::cin, tmp);

        if (!hList.insert(tmp))
            std::cout << "Failed" << std::endl;
    }

    if (action == 4) {

        std::string start;

        std::cout << "where to start? press enter to normalize from head" <<
std::endl;

        std::getline(std::cin, start);

        bool recursive = false;

        std::cout << "recursive? 1 - yes, 0 - no" << std::endl;

        std::cin >> recursive;

```

```

        hList.normalize(start, recursive);
    }
}

```

h_content.cpp :

```

#include "pch.h"
#include "h_content.h"

std::shared_ptr<Node> h_content::parse_str_into_node(std::string &source_str)
{
    std::string tmp_value_str = "";

    int tmp_index = 0;

    while (!isspace(source_str[tmp_index]))
    {
        if (source_str[tmp_index] != '.')
            tmp_value_str += source_str[tmp_index++];
        else {
            tmp_value_str = "";
            tmp_index++;
        }
    }

    int h_index_value = std::stoi(tmp_value_str);

    std::string title = "";
    std::string text = "";

    while ((tmp_index < source_str.size()) && (source_str[tmp_index] != '['))
        title += source_str[tmp_index++];

    if (tmp_index == source_str.size())
    {
        std::shared_ptr<Node> result(new Node);

        result->h_index = h_index_value;
        result->title = title;

        std::shared_ptr<Node> nullptr;
        result->list_or_text = nullptr;

        result->next = nullptr;

        return result;
    }
    else
    {
        while (tmp_index < source_str.size())
            text += source_str[tmp_index++];

        std::shared_ptr<Node> result(new Node);

        result->h_index = h_index_value;
        result->title = title;

        result->list_or_text = text;

        result->next = nullptr;
    }
}

```

```

        return result;
    }
}

void h_content::load_h_content(std::vector<std::string> &sorted_nodes)
{
    head = parse_str_into_node(sorted_nodes[0]);

    for (int i=1; i < sorted_nodes.size(); i++)
    {
        if (!add(sorted_nodes[i]))
            std::cout << "failed to add" << std::endl;
    }
}

std::shared_ptr<Node> h_content::insert(std::string &src_str)
{
    std::shared_ptr<Node> for_head_inserting;
    auto current_node = head;

    std::string tmp_value_str = "";

    int tmp_index = 0;

    while (!isspace(src_str[tmp_index]))
    {
        if (src_str[tmp_index] != '.')
            tmp_value_str += src_str[tmp_index++];
        else
        {
            int current_h_index = std::stoi(tmp_value_str);

            while (current_node->h_index != current_h_index) {
                if (current_node->next == nullptr)
                    return nullptr;
                current_node = current_node->next;
            }

            if (current_node->list_or_text.index() == 1)
                return nullptr;

            if (!std::get<0>(current_node->list_or_text))
                if ((std::find(src_str.begin() + tmp_index + 1,
src_str.end(), '.') < (std::find(src_str.begin() + tmp_index + 1, src_str.end(), ' ')))
                    return nullptr;
                else
                {
                    current_node->list_or_text =
parse_str_into_node(src_str);
                    return std::get<0>(current_node->list_or_text);
                }

            for_head_inserting = current_node;
            current_node = std::get<0>(current_node->list_or_text);

            tmp_index++;
            tmp_value_str = "";
        }
    }

    int current_h_index = std::stoi(tmp_value_str);

    while (current_node->h_index < current_h_index)
    {
        if (current_node->next) {

```

```

        if (current_node->next->h_index < current_h_index)
            current_node = current_node->next;
        else if (current_node->next->h_index == current_h_index) {
            auto tmp = current_node->next;

            current_node->next = parse_str_into_node(src_str);

            current_node->next->next = tmp;

            auto res = current_node->next;

            current_node = current_node->next->next;

            while (current_node->next)
            {
                current_node->h_index++;
                current_node = current_node->next;
            }

            current_node->h_index++;

            return res;
        }
        else if (current_node->next->h_index > current_h_index)
        {
            auto tmp = current_node->next;

            current_node->next = parse_str_into_node(src_str);

            current_node->next->next = tmp;

            return current_node->next;
        }
    }

    else
    {
        current_node->next = parse_str_into_node(src_str);

        return current_node->next;
    }
}

for_head_inserting->list_or_text = parse_str_into_node(src_str);

std::get<0>(for_head_inserting->list_or_text)->next = current_node;

return std::get<0>(for_head_inserting->list_or_text);
}

void h_content::print(std::shared_ptr<Node> head, std::string index_str)
{
    if (!head)
        return;

    std::cout << index_str + '.' + std::to_string(head->h_index) << " " << head->title
    << " ";

    if (head->list_or_text.index() == 1)
    {
        std::cout << std::get<1>(head->list_or_text) << std::endl;
        print(head->next, index_str);
    }
}

```

```

        else
        {
            std::cout << std::endl;
            print(std::get<0>(head->list_or_text), index_str + '.' +
std::to_string(head->h_index));
            print(head->next, index_str);
        }
    }

std::shared_ptr<Node> h_content::add(std::string &src_str)
{
    std::shared_ptr<Node> for_head_inserting;
    auto current_node = head;

    std::string tmp_value_str = "";

    int tmp_index = 0;

    while (!isspace(src_str[tmp_index]))
    {
        if (src_str[tmp_index] != '.')
            tmp_value_str += src_str[tmp_index++];
        else
        {
            int current_h_index = std::stoi(tmp_value_str);

            while (current_node->h_index != current_h_index) {
                if (current_node->next == nullptr)
                    return nullptr;
                current_node = current_node->next;
            }

            if (current_node->list_or_text.index() == 1)
                return nullptr;

            if (!std::get<0>(current_node->list_or_text))
                if ((std::find(src_str.begin() + tmp_index + 1,
src_str.end(), '.') < (std::find(src_str.begin() + tmp_index + 1, src_str.end(), ' ')))
                    return nullptr;
            else
            {
                current_node->list_or_text =
parse_str_into_node(src_str);
                return std::get<0>(current_node->list_or_text);
            }

            for_head_inserting = current_node;
            current_node = std::get<0>(current_node->list_or_text);

            tmp_index++;
            tmp_value_str = "";
        }
    }

    int current_h_index = std::stoi(tmp_value_str);

    while (current_node->h_index < current_h_index)
    {
        if (current_node->next) {
            if (current_node->next->h_index < current_h_index)
                current_node = current_node->next;
            else if (current_node->next->h_index == current_h_index)
                return nullptr;
            else if (current_node->next->h_index > current_h_index)

```

```

        {
            auto tmp = current_node->next;

            current_node->next = parse_str_into_node(src_str);

            current_node->next->next = tmp;

            return current_node->next;
        }
    }

    else
    {
        current_node->next = parse_str_into_node(src_str);

        return current_node->next;
    }
}

for_head_inserting->list_or_text = parse_str_into_node(src_str);

std::get<0>(for_head_inserting->list_or_text)->next = current_node;

return std::get<0>(for_head_inserting->list_or_text);
}

void h_content::normalize_row(std::shared_ptr<Node> start, bool recursive)
{
    if (!recursive) {
        int i = 0;

        while (start->next)
        {
            start->h_index = i++;
            start = start->next;
        }

        start->h_index = i;

        return;
    }
    else
    {
        int i = 0;

        while (start->next)
        {
            start->h_index = i++;

            if ((start->list_or_text.index() == 0) && (std::get<0>(start->list_or_text)))
                normalize_row(std::get<0>(start->list_or_text), true);

            start = start->next;
        }

        start->h_index = i;

        return;
    }
}

void h_content::normalize(const std::string &start, bool recursive)
{

```

```

if (!head)
{
    std::cout << "Failed" << std::endl;
    return;
}
if (!start.size())
{
    if (!recursive) {
        normalize_row(head);
    }

    else {
        normalize_row(head, recursive);
        return;
    }
}

else {
    auto current_node = head;

    std::string tmp_value_str = "";

    int tmp_index = 0;

    while (tmp_index < start.size())
    {
        if (start[tmp_index] != '.')
            tmp_value_str += start[tmp_index++];
        else
        {
            int current_h_index = std::stoi(tmp_value_str);

            while (current_node->h_index != current_h_index) {
                if (current_node->next == nullptr)
                {
                    std::cout << "Failed" << std::endl;
                    return;
                }
                current_node = current_node->next;
            }

            if (current_node->list_or_text.index() == 1)
            {
                std::cout << "Failed" << std::endl;
                return;
            }

            if (!std::get<0>(current_node->list_or_text))
                if ((std::find(start.begin() + tmp_index + 1,
start.end(), '.') < (std::find(start.begin() + tmp_index + 1, start.end(), ' ')))
                {
                    std::cout << "Failed" << std::endl;
                    return;
                }
            else
            {
                std::cout << "Failed" << std::endl;
                return;
            }

            current_node = std::get<0>(current_node->list_or_text);

            tmp_index++;
            tmp_value_str = "";
        }
    }
}

```



```

    }

    int current_h_index = std::stoi(tmp_value_str);

    while (current_node->h_index < current_h_index)
    {
        if (current_node->next) {
            if (current_node->next->h_index < current_h_index)
                current_node = current_node->next;
            else if (current_node->next->h_index == current_h_index)
            {
                auto tmp = head;

                head = std::get<0>(current_node->next->list_or_text);

                normalize("", recursive);

                head = tmp;
                return;
            }
            else if (current_node->next->h_index > current_h_index)
            {
                std::cout << "Failed" << std::endl;
                return;
            }
        }

        else
        {
            std::cout << "Failed" << std::endl;
            return;
        }
    }

    std::cout << "Failed" << std::endl;
    return;
}
}

```

h_content.h:

```

#pragma once
#include <string>
#include <variant>
#include <memory>
#include <vector>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <cctype>

typedef struct Node {
    int h_index; // "иерархический номер" (с)

    std::string title; // "название" (с)

    std::variant < std::shared_ptr<Node>, std::string > list_or_text; // "либо
несколько подчиненных записей, либо текст" (с)

    std::shared_ptr<Node> next;
}Node;

class h_content
{
public:

```

```

    h_content() = default;
    ~h_content() = default;

    std::shared_ptr<Node> head;

    void load_h_content(std::vector<std::string> &sorted_nodes); // forms a h_list,
sets head in h_content::head

    void print(std::shared_ptr<Node> head, std::string index_str);

    std::shared_ptr<Node> add(std::string &source_str);

    std::shared_ptr<Node> insert(std::string &source_str);

    void normalize(const std::string &start, bool recursive);
private:

    void normalize_row(std::shared_ptr<Node> start, bool recursive = false);

    std::shared_ptr<Node> parse_str_into_node(std::string &source_str);

};

```